# The Portable Executable File Format

## Abstract

The Windows NT™ version 3.1 operating system introduces a new executable file format called the Portable Executable (PE) file format.

## Introduction

The recent addition of the Microsoft® Windows NT™ operating system to the family of Windows™ operating systems brought many changes to the development environment and more than a few changes to applications themselves. One of the more significant changes is the introduction of the Portable Executable (PE) file format. The new PE file format draws primarily from the COFF (Common Object File Format) specification that is common to UNIX® operating systems. Yet, to remain compatible with previous versions of the MS-DOS® and Windows operating systems, the PE file format also retains the old familiar MZ header from MS-DOS.

This article discusses each of the components of the file as they occur when you traverse the file's contents, starting at the top and working your way down through the file.

Much of the definition of individual file components comes from the file WINNT.H, a file included in the Microsoft Win32™ Software Development Kit (SDK) for Windows NT. In it you will find structure type definitions for each of the file headers and data directories used to represent various components in the file. In other places in the file, WINNT.H lacks sufficient definition of the file structure.

The PE file format for Windows NT introduces a completely new structure to developers familiar with the Windows and MS-DOS environments. Yet developers familiar with the UNIX environment will find that the PE file format is similar to, if not based on, the COFF specification.

The entire format consists of an MS-DOS MZ header, followed by a real-mode stub program, the PE file signature, the PE file header, the PE optional header, all of the section headers, and finally, all of the section bodies.

The optional header ends with an array of data directory entries that are relative virtual addresses to data directories contained within section bodies. Each data directory indicates how a specific section body's data is structured.

The PE file format has eleven predefined sections, as is common to applications for Windows NT, but each application can define its own unique sections for code and data.

The .debug predefined section also has the capability of being stripped from the file into a separate debug file. If so, a special debug header is used to parse the debug file, and a flag is specified in the PE file header to indicate that the debug data has been stripped.

## Structure of PE Files

The PE file format is organized as a linear stream of data. It begins with an MS-DOS header, a real-mode program stub, and a PE file signature. Immediately following is a PE file header and optional header. Beyond that, all the section headers appear, followed by all of the section bodies. Closing out the file are a few other regions of miscellaneous information, including relocation information, symbol table information, line number information, and string table data. All of this is more easily absorbed by looking at it graphically, as shown in Figure 1.

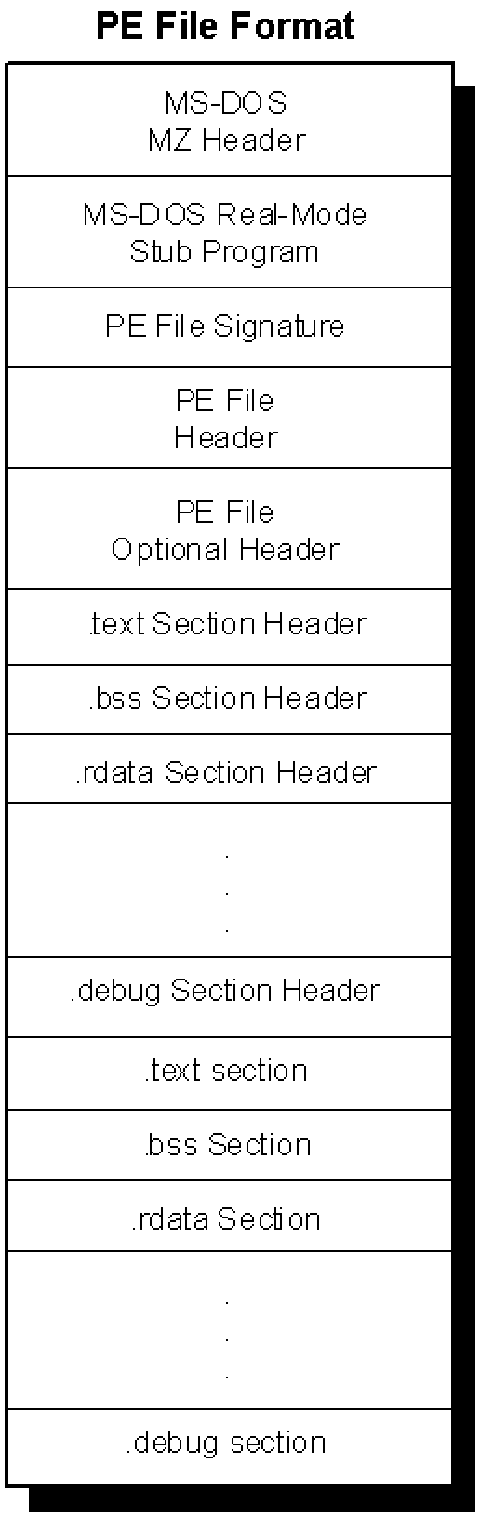

**Figure 1. Structure of a Portable Executable file image**

Starting with the MS-DOS file header structure, each of the components in the PE file format is discussed below in the order in which it occurs in the file.

**MS-DOS/Real-Mode Header**

As mentioned above, the first component in the PE file format is the MS-DOS header. The MS-DOS header is not new for the PE file format. It is the same MS-DOS header that has been around since version 2 of the MS-DOS operating system. The main reason for keeping the same structure intact at the beginning of the PE file format is so that, when you attempt to load a file created under Windows version 3.1 or earlier, or MS DOS version 2.0 or later, the operating system can read the file and understand that it is not compatible. In other words, when you attempt to run a Windows NT executable on MS-DOS version 6.0, you get this message: "This program cannot be run in DOS mode." If the MS-DOS header was not included as the first part of the PE file format, the operating system would simply fail the attempt to load the file and offer something completely useless, such as: "The name specified is not recognized as an internal or external command, operable program or batch file."

The MS-DOS header occupies the first 64 bytes of the PE file. A structure representing its content is described below:

**WINNT.H**

```
typedef struct _IMAGE_DOS_HEADER {  // DOS .EXE header
    USHORT e_magic;         // Magic number
    USHORT e_cblp;          // Bytes on last page of file
    USHORT e_cp;            // Pages in file
    USHORT e_crlc;          // Relocations
    USHORT e_cparhdr;       // Size of header in paragraphs
    USHORT e_minalloc;      // Minimum extra paragraphs needed
    USHORT e_maxalloc;      // Maximum extra paragraphs needed
    USHORT e_ss;            // Initial (relative) SS value
    USHORT e_sp;            // Initial SP value
    USHORT e_csum;          // Checksum
    USHORT e_ip;            // Initial IP value
    USHORT e_cs;            // Initial (relative) CS value
    USHORT e_lfarlc;        // File address of relocation table
    USHORT e_ovno;          // Overlay number
    USHORT e_res[4];        // Reserved words
    USHORT e_oemid;         // OEM identifier (for e_oeminfo)
    USHORT e_oeminfo;       // OEM information; e_oemid specific
    USHORT e_res2[10];      // Reserved words
    LONG   e_lfanew;        // File address of new exe header
  } IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

The first field, **e_magic**, is the so-called magic number. This field is used to identify an MS-DOS-compatible file type. All MS-DOS-compatible executable files set this value to 0x54AD, which represents the ASCII characters *MZ*. MS-DOS headers are sometimes referred to as MZ headers for this reason. Many other fields are important to MS-DOS operating systems, but for Windows NT, there is really one more important field in this structure. The final field, **e_lfanew**, is a 4-byte offset into the file where the PE file header is located. It is necessary to use this offset to locate the PE header in the file. For PE files in Windows NT, the PE file header occurs soon after the MS-DOS header with only the real-mode stub program between them.

### Real-Mode Stub Program

The real-mode stub program is an actual program run by MS-DOS when the executable is loaded. For an actual MS-DOS executable image file, the application begins executing here. For successive operating systems, including Windows, OS/2®, and Windows NT, an MS-DOS stub program is placed here that runs instead of the actual application. The programs typically do no more than output a line of text, such as: "This program requires Microsoft Windows v3.1 or greater." Of course, whoever creates the application is able to place any stub they like here, meaning you may often see such things as: "You can't run a Windows NT application on OS/2, it's simply not possible."

When building an application for Windows version 3.1, the linker links a default stub program called WINSTUB.EXE into your executable. You can override the default linker behavior by substituting your own valid MS-DOS-based program in place of WINSTUB and indicating this to the linker with the **STUB** module definition statement. Applications developed for Windows NT can do the same thing by using the **-STUB: linker** option when linking the executable file.

### PE File Header and Signature

The PE file header is lo cated by indexing the **e_lfanew** field of the MS-DOS header. The **e_lfanew** field simply gives the offset in the file, so add the file's memory-mapped base address to determine the actual memory-mapped address

```
#define NTSIGNATURE(a) ((LPVOID)((BYTE *)a +     \
                        ((PIMAGE_DOS_HEADER)a)->e_lfanew))
```

When manipulating PE file information, I found that there were several locations in the file that I needed to refer to often. Since these locations are merely offsets into the file, it is easier to implement these locations as macros because they provide much better performance than functions do.

Notice that instead of retrieving the offset of the PE file header, this macro retrieves the location of the PE file signature. Starting with Windows and OS/2 executables, .EXE files were given file signatures to specify the intended target operating system. For the PE file format in Windows NT, this signature occurs immediately before the PE file header structure. In versions of Windows and OS/2, the signature is the first word of the file header. Also, for the PE file format, Windows NT uses a DWORD for the signature.

The macro presented above returns the offset of where the file signature appears, regardless of which type of executable file it is. So depending on whether it's a Windows NT file signature or not, the file header exists either after the signature DWORD or at the signature WORD.

```
DWORD  WINAPI ImageFileType (
    LPVOID    lpFile)
{
    /* DOS file signature comes first. */
    if (*(USHORT *)lpFile == IMAGE_DOS_SIGNATURE)
        {
        /* Determine location of PE File header from
           DOS header. */
        if (LOWORD (*(DWORD *)NTSIGNATURE (lpFile)) ==
                               IMAGE_OS2_SIGNATURE ||
            LOWORD (*(DWORD *)NTSIGNATURE (lpFile)) ==
                            IMAGE_OS2_SIGNATURE_LE)
            return (DWORD)LOWORD(*(DWORD *)NTSIGNATURE (lpFile));

        else if (*(DWORD *)NTSIGNATURE (lpFile) ==
                           IMAGE_NT_SIGNATURE)
            return IMAGE_NT_SIGNATURE;

        else
            return IMAGE_DOS_SIGNATURE;
        }

    else
        /* unknown file type */
        return 0;
}
```

The code listed above quickly shows how useful the **NTSIGNATURE** macro becomes. The macro makes it easy to compare the different file types and return the appropriate one for a given type of file. The four different file types defined in WINNT.H are:

**WINNT.H**

```
#define IMAGE_DOS_SIGNATURE             0x5A4D      // MZ
#define IMAGE_OS2_SIGNATURE             0x454E      // NE
#define IMAGE_OS2_SIGNATURE_LE          0x454C      // LE
#define IMAGE_NT_SIGNATURE              0x00004550  // PE00
```

At first it seems curious that Windows executable file types do not appear on this list. But then, after a little investigation, the reason becomes clear: There really is no difference between Windows executables and OS/2 executables other than the operating system version specification. Both operating systems share the same executable file structure.

Turning our attention back to the Windows NT PE file format, we find that once we have the location of the file signature, the PE file follows four bytes later. The next macro identifies the PE file header:

```
#define PEFHDROFFSET(a) ((LPVOID)((BYTE *)a +  \
    ((PIMAGE_DOS_HEADER)a)->e_lfanew + SIZE_OF_NT_SIGNATURE))
```

The only difference between this and the previous macro is that this one adds in the constant SIZE_OF_NT_SIGNATURE. Sad to say, this constant is not defined in WINNT.H.

Now that we know the location of the PE file header, we can examine the data in the header simply by assigning this location to a structure, as in the following example:

```
PIMAGE_FILE_HEADER   pfh;

pfh = (PIMAGE_FILE_HEADER)PEFHDROFFSET (lpFile);
```

In this example, *lpFile* represents a pointer to the base of the memory-mapped executable file, and therein lies the convenience of memory-mapped files. No file I/O needs to be performed; simply dereference the pointer *pfh* to access information in the file. The PE file header structure is defined as:

**WINNT.H**

```
typedef struct _IMAGE_FILE_HEADER {
    USHORT  Machine;
    USHORT  NumberOfSections;
    ULONG   TimeDateStamp;
    ULONG   PointerToSymbolTable;
    ULONG   NumberOfSymbols;
    USHORT  SizeOfOptionalHeader;
    USHORT  Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;

#define IMAGE_SIZEOF_FILE_HEADER             20
```

Notice that the size of the file header structure is conveniently defined in the include file...

The information in the PE file is basically high-level information that is used by the system or applications to determine how to treat the file. The first field is used to indicate what type of machine the executable was built for, such as the DEC® Alpha, MIPS R4000, Intel® x86, or some other processor. The system uses this information to quickly determine how to treat the file before going any further into the rest of the file data.

The *Characteristics* field identifies specific characteristics about the file. For example, consider how separate debug files are managed for an executable. It is possible to strip debug information from a PE file and store it in a debug file (.DBG) for use by debuggers. To do this, a debugger needs to know whether to find the debug information in a separate file or not and whether the information has been stripped from the file or not. A debugger could find out by drilling down into the executable file looking for debug information. To save the debugger from having to search the file, a file characteristic that indicates that the file has been stripped (IMAGE_FILE_DEBUG_STRIPPED) was invented. Debuggers can look in the PE file header to quickly determine whether the debug information is present in the file or not.

WINNT.H defines several other flags that indicate file header information much the way the example described above does. I'll leave it as an exercise for the reader t o look up the flags to see if any of them are interesting or not. They are located in WINNT.H immediately after the **IMAGE_FILE_HEADER** structure described above.

One other useful entry in the PE file header structure is the *NumberOfSections* field. It turns out that you need to know how many sections--more specifically, how many section headers and section bodies--are in the file in order to extract the information easily. Each section header and section body is laid out sequentially in the file, so the number of sections is necessary to determine where the section headers and bodies end. The following function extracts the number of sections from the PE file header:

```
int   WINAPI NumOfSections (
    LPVOID    lpFile)
{
    /* Number of sections is indicated in file header. */
    return (int)((PIMAGE_FILE_HEADER)
                  PEFHDROFFSET (lpFile))->NumberOfSections);
}
```

As you can see, the **PEFHDROFFSET** and the other macros are pretty handy to have around.

**PE Optional Header**

The next 224 bytes in the executable file make up the PE optional header. Though its name is "optional header," rest assured that this is not an optional entry in PE executable files. A pointer to the optional header is obtained with the **OPTHDROFFSET** macro:

```
#define OPTHDROFFSET(a) ((LPVOID)((BYTE *)a                 + \
    ((PIMAGE_DOS_HEADER)a)->e_lfanew + SIZE_OF_NT_SIGNATURE + \
    sizeof (IMAGE_FILE_HEADER)))
```

The optional header contains most of the meaningful information about the executable image, such as initial stack size, program entry point location, preferred base address, operating system version, section alignment information, and so forth. The **IMAGE_OPTIONAL_HEADER** structure represents the optional header as follows:

**WINNT.H**

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    //
    // Standard fields.
    //
    USHORT  Magic;
    UCHAR   MajorLinkerVersion;
    UCHAR   MinorLinkerVersion;
    ULONG   SizeOfCode;
    ULONG   SizeOfInitializedData;
    ULONG   SizeOfUninitializedData;
    ULONG   AddressOfEntryPoint;
    ULONG   BaseOfCode;
    ULONG   BaseOfData;
    //
    // NT additional fields.
    //
    ULONG   ImageBase;
    ULONG   SectionAlignment;
    ULONG   FileAlignment;
    USHORT  MajorOperatingSystemVersion;
    USHORT  MinorOperatingSystemVersion;
    USHORT  MajorImageVersion;
    USHORT  MinorImageVersion;
    USHORT  MajorSubsystemVersion;
    USHORT  MinorSubsystemVersion;
    ULONG   Reserved1;
    ULONG   SizeOfImage;
    ULONG   SizeOfHeaders;
    ULONG   CheckSum;
    USHORT  Subsystem;
    USHORT  DllCharacteristics;
    ULONG   SizeOfStackReserve;
    ULONG   SizeOfStackCommit;
    ULONG   SizeOfHeapReserve;
    ULONG   SizeOfHeapCommit;
    ULONG   LoaderFlags;
    ULONG   NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;
```

As you can see, the list of fields in this structure is rather lengthy. Rather than bore you with descriptions of all of these fields, I'll simply discuss the useful ones--that is, useful in the context of exploring the PE file format.

**Standard Fields**

First, note that the structure is divided into "Standard fields" and "NT additional fields." The standard fields are those common to the Common Object File Format (COFF), which most UNIX executable files use. Though the standard fields retain the names defined in COFF, Windows NT actually uses some of them for different purposes that would be better described with other names.

- *Magic*. I was unable to track down what this field is used for.
- *MajorLinkerVersion*, *MinorLinkerVersion*. Indicates version of the linker that linked this image.
- *SizeOfCode*. Size of executable code.
- *SizeOfInitializedData*. Size of initialized data.
- *SizeOfUninitializedData*. Size of uninitialized data.
- *AddressOfEntryPoint*. Of the standard fields, the *AddressOfEntryPoint* field is the most interesting for the PE file format. This field indicates the location of the entry point for the application and, perhaps more importantly to system hackers, the location of the end of the Import Address Table (IAT). The following function demonstrates how to retrieve the entry point of a Windows NT executable image from the optional header.

```
        LPVOID  WINAPI GetModuleEntryPoint (
            LPVOID    lpFile)
        {
            PIMAGE_OPTIONAL_HEADER   poh;

            poh = (PIMAGE_OPTIONAL_HEADER)OPTHDROFFSET (lpFile);

            if (poh != NULL)
                return (LPVOID)poh->AddressOfEntryPoint;
            else
```

```
                return NULL;
        }
```

- *BaseOfCode*. Relative offset of code (".text" section) in loaded image.
- *BaseOfData*. Relative offset of uninitialized data (".bss" section) in loaded image.

**Windows NT Additional Fields**

The additional fields added to the Windows NT PE file format provide loader support for much of the Windows NT-specific process behavior. Following is a summary of these fields.

- *ImageBase*. Preferred base address in the address space of a process to map the executable image to. The linker defaults to 0x00400000, but you can override the default with the **-BASE: linker** switch.
- *SectionAlignment*. Each section is loaded into the address space of a process sequentially, beginning at *ImageBase*. *SectionAlignment* dictates the minimum amount of space a section can occupy when loaded--that is, sections are aligned on *SectionAlignment* boundaries.

  Section alignment can be no less than the page size (currently 4096 bytes on the *x*86 platform) and must be a multiple of the page size as dictated by the behavior of Windows NT's virtual memory manager. 4096 bytes is the *x*86 linker default, but this can be set using the **-ALIGN: linker** switch.

- *FileAlignment*. Minimum granularity of chunks of information within the image file prior to loading. For example, the linker zero-pads a section body (raw data for a section) up to the nearest *FileAlignment* boundary in the file. This value is constrained to be a power of 2 between 512 and 65,535.
- *MajorOperatingSystemVersion*. Indicates the major version of the Windows NT operating system.
- *MinorOperatingSystemVersion*. Indicates the minor version of the Windows NT operating system.
- *MajorImageVersion*. Used to indicate the major version number of the application.
- *MinorImageVersion*. Used to indicate the minor version number of the application.
- *MajorSubsystemVersion*. Indicates the Windows NT Win32 subsystem major version number.
- *MinorSubsystemVersion*. Indicates the Windows NT Win32 subsystem minor version number.
- *Reserved1*. Unknown purpose, currently not used by the system and set to zero by the linker.
- *SizeOfImage*. Indicates the amount of address space to reserve in the address space for the loaded executable image. This number is influenced greatly by *SectionAlignment*. For example, consider a system having a fixed page size of 4096 bytes. If you have an executable with 11 sections, each less than 4096 bytes, aligned on a 65,536-byte boundary, the *SizeOfImage* field would be set to 11 * 65,536 = 720,896 (176 pages). The same file linked with 4096-byte alignment would result in 11 * 4096 = 45,056 (11 pages) for the *SizeOfImage* field. This is a simple example in which each section requires less than a page of memory. In reality, the linker determines the exact *SizeOfImage* by figuring each section individually. It first determines how many bytes the section requires, then it rounds up to the nearest page boundary, and finally it rounds page count to the nearest *SectionAlignment* boundary. The total is then the sum of each section's individual requirement.
- *SizeOfHeaders*. This field indicates how much space in the file is used for representing all the file headers, including the MS-DOS header, PE file header, PE optional header, and PE section headers. The section bodies begin at this location in the file.
- *CheckSum*. A checksum value is used to validate the executable file at load time. The value is set and verified by the linker. The algorithm used for creating these checksum values is proprietary information and will not be published.
- *Subsystem*. Field used to identify the target subsystem for this executable. Each of the possible subsystem values are listed in the WINNT.H file immediately after the **IMAGE_OPTIONAL_HEADER** structure.
- *DllCharacteristics*. Flags used to indicate if a DLL image includes entry points for process and thread initialization and termination.
- *SizeOfStackReserve*, *SizeOfStackCommit*, *SizeOfHeapReserve*, *S izeOfHeapCommit*. These fields control the amount of address space to reserve and commit for the stack and default heap. Both the stack and heap have default values of 1 page committed and 16 pages reserved. These values are set with the linker switches **-STACKSIZE:** and **-HEAPSIZE:**.
- *LoaderFlags*. Tells the loader whether to break on load, debug on load, or the default, which is to let things run normally.
- *NumberOfRvaAndSizes*. This field identifies the length of the *DataDirectory* array that follows. It is important to note that this field is used to identify the size of the array, not the number of valid entries in the array.
- *DataDirectory*. The data directory indicates where to find other important components of executable information in the file. It is really nothing more than an array of **IMAGE_DATA_DIRECTORY** structures that are located at the end of the optional header structure. The current PE file format defines 16 possible data directories, 11 of which are now being used.

**Data Directories**

As defined in WINNT.H, the data directories are:

**WINNT.H**

```
// Directory Entries

// Export Directory
#define IMAGE_DIRECTORY_ENTRY_EXPORT         0
// Import Directory
#define IMAGE_DIRECTORY_ENTRY_IMPORT         1
// Resource Directory
#define IMAGE_DIRECTORY_ENTRY_RESOURCE       2
// Exception Directory
#define IMAGE_DIRECTORY_ENTRY_EXCEPTION      3
// Security Directory
#define IMAGE_DIRECTORY_ENTRY_SECURITY       4
// Base Relocation Table
#define IMAGE_DIRECTORY_ENTRY_BASERELOC      5
// Debug Directory
#define IMAGE_DIRECTORY_ENTRY_DEBUG          6
// Description String
#define IMAGE_DIRECTORY_ENTRY_COPYRIGHT      7
// Machine Value (MIPS GP)
#define IMAGE_DIRECTORY_ENTRY_GLOBALPTR      8
// TLS Directory
#define IMAGE_DIRECTORY_ENTRY_TLS            9
// Load Configuration Directory
#define IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG    10
```

Each data directory is basically a structure defined as an **IMAGE_DATA_DIRECTORY**. And although data directory entries themselves are the same, each specific directory type is entirely unique. The definition of each defined data directory is described in "Predefined Sections" later in this article.

**WINNT.H**

```
typedef struct _IMAGE_DATA_DIRECTORY {
    ULONG   VirtualAddress;
    ULONG   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

Each data directory entry specifies the size and relative virtual address of the directory. To locate a particular directory, you determine the relative address from the data directory array in the optional header. Then use the virtual address to determine which section the directory is in. Once you determine which section contains the directory, the section header for that section is then used to find the exact file offset location of the data directory.

So to get a data directory, you first need to know about sections, which are described next. An example of how to locate data directories immediately follows this discussion.

## PE File Sections

The PE file specification consists of the headers defined so far and a generic object called a *section*. Sections contain the content of the file, including code, data, resources, and other executable information. Each section has a header and a body (the raw data). Section headers are described below, but section bodies lack a rigid file structure. They can be organized in almost any way a linker wishes to organize them, as long as the header is filled with enough information to be able to decipher the data.

### Section Headers

Section headers are located sequentially right after the optional header in the PE file format. Each section header is 40 bytes with no padding between them. Section headers are defined as in the following structure:

**WINNT.H**

```
#define IMAGE_SIZEOF_SHORT_NAME              8

typedef struct _IMAGE_SECTION_HEADER {
    UCHAR   Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
            ULONG   PhysicalAddress;
            ULONG   VirtualSize;
    } Misc;
    ULONG   VirtualAddress;
    ULONG   SizeOfRawData;
    ULONG   PointerToRawData;
    ULONG   PointerToRelocations;
    ULONG   PointerToLinenumbers;
    USHORT  NumberOfRelocations;
    USHORT  NumberOfLinenumbers;
    ULONG   Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

How do you go about getting section header information for a particular section? Since section headers are organized sequentially in no specific order, section headers must be located by name. The following function shows how to retrieve a section header from a PE image file given the name of the section:

```
BOOL    WINAPI GetSectionHdrByName (
    LPVOID                   lpFile,
    IMAGE_SECTION_HEADER     *sh,
    char                     *szSection)
{
    PIMAGE_SECTION_HEADER    psh;
    int                      nSections = NumOfSections (lpFile);
    int                      i;


    if ((psh = (PIMAGE_SECTION_HEADER)SECHDROFFSET (lpFile)) !=
         NULL)
        {
        /* find the section by name */
        for (i=0; i<nSections; i++)
            {
            if (!strcmp (psh->Name, szSection))
                {
                /* copy data to header */
                CopyMemory ((LPVOID)sh,
                            (LPVOID)psh,
                            sizeof (IMAGE_SECTION_HEADER));
                return TRUE;
                }
            else
                psh++;
            }
        }

    return FALSE;
}
```

The function simply locates the first section header via the **SECHDROFFSET** macro. Then the function loops through each section, comparing each section's name with the name of the section it's looking for, until it finds the right one. When the section is found, the function copies the data from the memory-mapped file to the structure passed in to the function. The fields of the **IMAGE_SECTION_HEADER** structure can then be accessed directly from the structure.

**Section Header Fields**

- *Name*. Each section header has a *name* field up to eight characters long, for which the first character must be a period.
- *PhysicalAddress* or *VirtualSize*. The second field is a union field that is not currently used.
- *VirtualAddress*. This field identifies the virtual address in the process's address space to which to load the section. The actual address is created by taking the value of this field and adding it to the *ImageBase* virtual address in the optional header structure. Keep in mind, though, that if this image file represents a DLL, there is no guarantee that the DLL will be loaded to the *ImageBase* location requested. So once the file is loaded into a process, the actual *ImageBase* value should be verified programmatically using **GetModuleHandle**.
- *SizeOfRawData*. This field indicates the *FileAlignment*-relative size of the section body. The actual size of the section body will be less than or equal to a multiple of *FileAlignment* in the file. Once the image is loaded into a process's address space, the

size of the section body becomes less than or equal to a multiple of *SectionAlignment*.

- *PointerToRawData*. This is an offset to the location of the section body in the file.
- *PointerToRelocations*, *PointerToLinenumbers*, *NumberOfRelocations*, *NumberOfLinenumbers*. None of these fi elds are used in the PE file format.
- *Characteristics*. Defines the section characteristics. These values are found both in WINNT.H and in the Portable Executable Format specification located on this CD.

| **Value** | **Definition** |
|---|---|
| | |
| 0x00000020 | Code section |
| | |
| 0x00000040 | Initialized data section |
| | |
| 0x00000080 | Uninitialized data section |
| | |
| 0x04000000 | Section cannot be cached |
| | |
| 0x08000000 | Section is not pageable |
| | |
| 0x10000000 | Section is shared |
| | |
| 0x20000000 | Executable section |
| | |
| 0x40000000 | Readable section |
| | |
| 0x80000000 | Writable section |

**Locating Data Directories**

Data directories exist within the body of their corresponding data section. Typically, data directories are the first structure within the section body, but not out of necessity. For that reason, you need to retrieve information from both the section header and optional header to locate a specific data directory.

To make this process easier, the following function was written to locate the data directory for any of the directories defined in WINNT.H:

```
LPVOID  WINAPI ImageDirectoryOffset (
        LPVOID    lpFile,
        DWORD     dwIMAGE_DIRECTORY)
{
    PIMAGE_OPTIONAL_HEADER   poh;
    PIMAGE_SECTION_HEADER    psh;
    int                      nSections = NumOfSections (lpFile);
    int                      i = 0;
    LPVOID                   VAImageDir;

    /* Must be 0 thru (NumberOfRvaAndSizes-1). */
    if (dwIMAGE_DIRECTORY >= poh->NumberOfRvaAndSizes)
        return NULL;
    /* Retrieve offsets to optional and section headers. */
    poh = (PIMAGE_OPTIONAL_HEADER)OPTHDROFFSET (lpFile);
    psh = (PIMAGE_SECTION_HEADER)SECHDROFFSET (lpFile);

    /* Locate image directory's relative virtual address. */
    VAImageDir = (LPVOID)poh->DataDirectory
                       [dwIMAGE_DIRECTORY].VirtualAddress;

    /* Locate section containing image directory. */
    while (i++<nSections)
        {
        if (psh->VirtualAddress <= (DWORD)VAImageDir &&
            psh->VirtualAddress +
                 psh->SizeOfRawData > (DWORD)VAImageDir)
            break;
        psh++;
        }

    if (i > nSections)
        return NULL;

    /* Return image import directory offset. */
    return (LPVOID)(((int)lpFile +
                     (int)VAImageDir. psh->VirtualAddress) +
                    (int)psh->PointerToRawData);
}
```

The function begins by validating the requested data directory entry number. Then it retrieves pointers to the optional header and first section header. From the optional header, the function determines the data directory's virtual address, and it uses this value to determine within which section body the data directory is located. Once the appropriate section body has been identified, the specific location of the data directory is found by translating the relative virtual address of the data directory to a specific address into the file.

**Predefined Sections**

An application for Windows NT typically has the nine predefined sections named .text, .bss, .rdata, .data, .rsrc, .edata, .idata, .pdata, and .debug. Some applications do not need all of these sections, while others may define still more sections to suit their specific needs. This behavior is similar to code and data segments in MS-DOS and Windows version 3.1. In fact, the way an application defines a unique section is by using the standard compiler directives for naming code and data segments or by using the name segment compiler option **-NT**--exactly the same way in which applications defined unique code and data segments in Windows version 3.1.

The following is a discussion of some of the more interesting sections common to typical Windows NT PE files.

**Executable code section, .text**

One difference between Windows version 3.1 and Windows NT is that the default behavior combines all code segments (as they are referred to in Windows version 3.1) into a single section called ".text" in Windows NT. Since Windows NT uses a page-based virtual memory management system, there is no advantage to separating code into distinct code segments. Consequently, having one large code section is easier to manage for both the operating system and the application developer.

The .text section also contains the entry point mentioned earlier. The IAT also lives in the .text section immediately before the module entry point. (The IAT's presence in the .text section makes sense because the table is really a series of jump instructions, for which the specific location to jump to is the fixed-up address.) When Windows NT executable images are loaded into a process's address space, the IAT is fixed up with the location of each imported function's physical address. In order to find the IAT in the .text section, the loader simply locates the module entry point and relies on the fact that the IAT occurs immediately before the entry point. And since each entry is the same size, it is easy to walk backward in the table to find its beginning.

**Data sections, .bss, .rdata, .data**

The .bss section represents uninitialized data for the application, including all variables declared as static within a function or source module.

The .rdata section represents read-only data, such as literal strings, constants, and debug directory information.

All other variables (except automatic variables, which appear on the stack) are stored in the .data section. Basically, these are application or module global variables.

**Resources section, .rsrc**

The .rsrc section contains resource information for a module. It begins with a resource directory structure like most other sections, but this section's data is further structured into a resource tree. The **IMAGE_RESOURCE_DIRECTORY**, shown below, forms the root and nodes of the tree.

**WINNT.H**

```
typedef struct _IMAGE_RESOURCE_DIRECTORY {
    ULONG   Characteristics;
    ULONG   TimeDateStamp;
    USHORT  MajorVersion;
    USHORT  MinorVersion;
    USHORT  NumberOfNamedEntries;
    USHORT  NumberOfIdEntries;
} IMAGE_RESOURCE_DIRECTORY, *PIMAGE_RESOURCE_DIRECTORY;
```

Looking at the directory structure, you won't find any pointer to the next nodes. Instead, there are two fields, *NumberOfNamedEntries* and *NumberOfIdEntries*, used to indicate how many entries are attached to the directory. By *attached*, I mean the directory entries follow immediately after the directory in the section data. The named entries appear first in ascending alphabetical order, followed by the ID entries in ascending numerical order.

A directory entry consists of two fields, as described in the following **IMAGE_RESOURCE_DIRECTORY_ENTRY** structure:

**WINNT.H**

```
typedef struct _IMAGE_RESOURCE_DIRECTORY_ENTRY {
    ULONG   Name;
    ULONG   OffsetToData;
} IMAGE_RESOURCE_DIRECTORY_ENTRY, *PIMAGE_RESOURCE_DIRECTORY_ENTRY;
```

The two fields are used for different things depending on the level of the tree. The *Name* field is used to identify either a type of resource, a resource name, or a resource's language ID. The *OffsetToData* field is always used to point to a sibling in the tree, either a directory node or a leaf node.

Leaf nodes are the lowest node in the resource tree. They define the size and location of the actual resource data. Each leaf node is represented using the following **IMAGE_RESOURCE_DATA_ENTRY** structure:

**WINNT.H**

```
typedef struct _IMAGE_RESOURCE_DATA_ENTRY {
    ULONG   OffsetToData;
    ULONG   Size;
    ULONG   CodePage;
    ULONG   Reserved;
} IMAGE_RESOURCE_DATA_ENTRY, *PIMAGE_RESOURCE_DATA_ENTRY;
```

The two fields *OffsetToData* and *Size* indicate the location and size of the actual resource data. Since this information is used primarily by functions once the application has been loaded, it makes more sense to make the *OffsetToData* field a relative virtual address. This is precisely the case. Interestingly enough, all other offsets, such as pointers from directory entries to other directories, are offsets relative to the location of the root node.

To make all of this a little clearer, consider Figure 2.

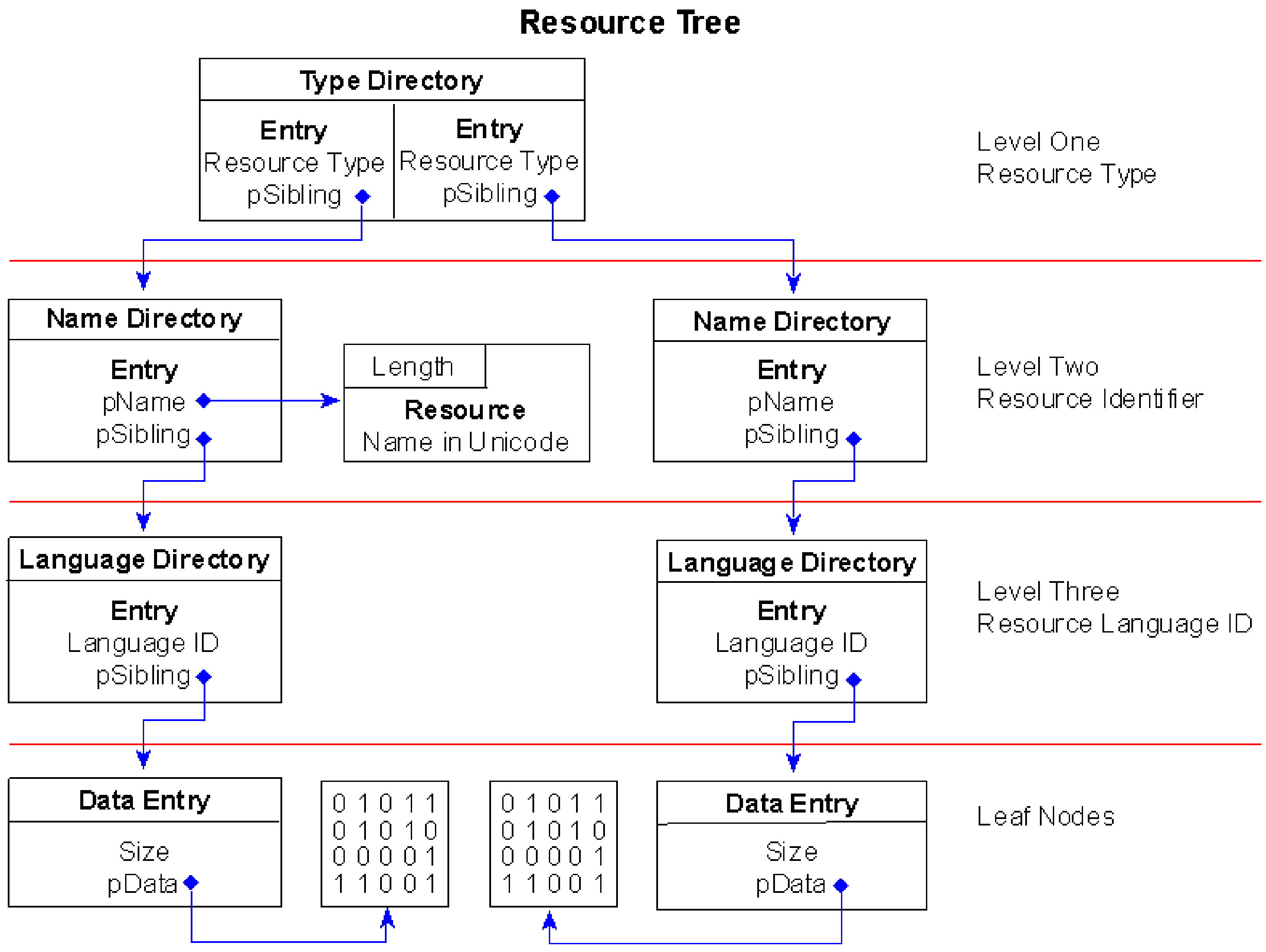

**Figure 2. A simple resource tree structure**

Figure 2 depicts a very simple resource tree containing only two resource objects, a menu, and a string table. Further, the menu and string table have only one item each. Yet, you can see how complicated the resource tree becomes--even with as few resources as this.

At the root of the tree, the first directory has one entry for each type of resource the file contains, no matter how many of each type there are. In Figure 2, there are two entries identified by the root, one for the menu and one for the string table. If there had been one or more dialog resources included in the file, the root node would have had one more entry and, consequently, another branch for the dialog resources.

The basic resource types are identified in the file WINUSER.H and are listed below:

**WINUSER.H**

```
/*
 * Predefined Resource Types
 */
#define RT_CURSOR           MAKEINTRESOURCE(1)
#define RT_BITMAP           MAKEINTRESOURCE(2)
#define RT_ICON             MAKEINTRESOURCE(3)
#define RT_MENU             MAKEINTRESOURCE(4)
#define RT_DIALOG           MAKEINTRESOURCE(5)
#define RT_STRING           MAKEINTRESOURCE(6)
#define RT_FONTDIR          MAKEINTRESOURCE(7)
#define RT_FONT             MAKEINTRESOURCE(8)
#define RT_ACCELERATOR      MAKEINTRESOURCE(9)
#define RT_RCDATA           MAKEINTRESOURCE(10)
```

```
#define RT_MESSAGETABLE      MAKEINTRESOURCE(11)
```

At the top level of the tree, the MAKEINTRESOURCE values listed above are placed in the *Name* field of each type entry, identifying the different resources by type.

Each of the entries in the root directory points to a sibling node in the second level of the tree. These nodes are directories, too, each having their own entries. At this level, the directories are used to identify the name of each resource within a given type. If you had multiple menus defined in your application, there would be an entry for each one here at the second level of the tree.

As you are probably already aware, resources can be identified by name or by integer. They are distinguished in this level of the tree via the *Name* field in the directory structure. If the most significant bit of the *Name* field is set, the other 31 bits are used as an offset to an **IMAGE_RESOURCE_DIR_STRING_U** structure.

**WINNT.H**

```
typedef struct _IMAGE_RESOURCE_DIR_STRING_U {
    USHORT  Length;
    WCHAR   NameString[ 1 ];
} IMAGE_RESOURCE_DIR_STRING_U, *PIMAGE_RESOURCE_DIR_STRING_U;
```

This structure is simply a 2-byte *Length* field followed by *Length* UNICODE characters.

On the other hand, if the most significant bit of the *Name* field is clear, the lower 31 bits are used to represent the integer ID of the resource. Figure 2 shows the menu resource as a named resource and the string table as an ID resource.

If there were two menu resources, one identified by name and one by resource, they would both have entries immediately after the menu resource directory. The named resource entry would appear first, followed by the integer-identified resource. The directory fields *NumberOfNamedEntries* and *NumberOfIdEntries* would each contain the value 1, indicating the presence of one entry.

Below level two, the resource tree does not branch out any further. Level one branches into directories representing each type of resource, and level two branches into directories representing each resource by identifier. Level three maps a one-to-one correspondence between the individually identified resources and their respective language IDs. To indicate the language ID of a resource, the *Name* field of the directory entry structure is used to indicate both the primary language and sublanguage ID for the resource. For the value 0x0409, 0x09 represents the primary language as LANG_ENGLISH, and 0x04 is defined as SUBLANG_ENGLISH_CAN for the sublanguage. The entire set of language IDs is defined in the file WINNT.H.

Since the language ID node is the last directory node in the tree, the *OffsetToData* field in the entry structure is an offset to a leaf node--the **IMAGE_RESOURCE_DATA_ENTRY** structure mentioned earlier.

Referring back to Figure 2, you can see one data entry node for each language directory entry. This node simply indicates the size of the resource data and the relative virtual address where the resource data is located.

One advantage to having so much structure to the resource data section, .rsrc, is that you can glean a great deal of information from the section without accessing the resources themselves. For example, you can find out how many there are of each type of resource, what resources--if any--use a particular language ID, whether a particular resource exists or not, and the size of individual types of resources. To demonstrate how to make use of this information, the following function shows how to determine the different types of resources a file includes:

```
int     WINAPI GetListOfResourceTypes (
    LPVOID    lpFile,
    HANDLE    hHeap,
    char      **pszResTypes)
{
    PIMAGE_RESOURCE_DIRECTORY          prdRoot;
    PIMAGE_RESOURCE_DIRECTORY_ENTRY    prde;
    char                               *pMem;
    int                                nCnt, i;


    /* Get root directory of resource tree. */
    if ((prdRoot = PIMAGE_RESOURCE_DIRECTORY)ImageDirectoryOffset
           (lpFile, IMAGE_DIRECTORY_ENTRY_RESOURCE)) == NULL)
        return 0;

    /* Allocate enough space from heap to cover all types. */
    nCnt = prdRoot->NumberOfIdEntries * (MAXRESOURCENAME + 1);
    *pszResTypes = (char *)HeapAlloc (hHeap,
                                              HEAP_ZERO_MEMORY,
                                              nCnt);
    if ((pMem = *pszResTypes) == NULL)
        return 0;

    /* Set pointer to first resource type entry. */
    prde = (PIMAGE_RESOURCE_DIRECTORY_ENTRY)((DWORD)prdRoot +
               sizeof (IMAGE_RESOURCE_DIRECTORY));

    /* Loop through all resource directory entry types. */
    for (i=0; i<prdRoot->NumberOfIdEntries; i++)
        {
        if (LoadString (hDll, prde->Name, pMem, MAXRESOURCENAME))
            pMem += strlen (pMem) + 1;

        prde++;
        }

    return nCnt;
}
```

This function returns a list of resource type names in the string identified by *pszResTypes*. Notice that, at the heart of this function, **LoadString** is called using the *Name* field of each resource type directory entry as the string ID.of resource type strings whose IDs are defined the same as the type specifiers in the directory entries.It would be rather easy to expand on these functions or write new functions that extracted other information from this section.

**Export data section, .edata**

The .edata section contains export data for an application or DLL. When present, this section contains an export directory for getting to the export information.

**WINNT.H**

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    ULONG   Characteristics;
    ULONG   TimeDateStamp;
    USHORT  MajorVersion;
    USHORT  MinorVersion;
    ULONG   Name;
    ULONG   Base;
    ULONG   NumberOfFunctions;
    ULONG   NumberOfNames;
    PULONG  *AddressOfFunctions;
    PULONG  *AddressOfNames;
    PUSHORT *AddressOfNameOrdinals;
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

The *Name* field in the export directory identifies the name of the executable module. *NumberOfFunctions* and *NumberOfNames* fields indicate how many functions and function names are being exported from the module.

The *AddressOfFunctions* field is an offset to a list of exported function entry points. The *AddressOfNames* field is the address of an offset to the beginning of a null-separated list of exported function names. *AddressOfNameOrdinals* is an offset to a list of ordinal values (each 2 bytes long) for the same exported functions.

The three *AddressOf...* fields are relative virtual addresses into the address space of a process once the module has been loaded. Once the module is loaded, the relative virtual address should be added to the module base address to get the exact location in the address space of the process. Before the file is loaded, however, the address can be determined by subtracting the section header virtual address (*VirtualAddress*) from the given field address, adding the section body offset (*PointerToRawData*) to the result, and then using this value as an offset into the image file. The following example illustrates this technique:

```
int  WINAPI GetExportFunctionNames (
    LPVOID    lpFile,
    HANDLE    hHeap,
    char      **pszFunctions)
{
    IMAGE_SECTION_HEADER       sh;
    PIMAGE_EXPORT_DIRECTORY    ped;
    char                       *pNames, *pCnt;
    int                        i, nCnt;

    /* Get section header and pointer to data directory
       for .edata section. */
    if ((ped = (PIMAGE_EXPORT_DIRECTORY)ImageDirectoryOffset
            (lpFile, IMAGE_DIRECTORY_ENTRY_EXPORT)) == NULL)
        return 0;
    GetSectionHdrByName (lpFile, &sh, ".edata");

    /* Determine the offset of the export function names. */
    pNames = (char *)(*(int *)((int)ped->AddressOfNames -
                               (int)sh.VirtualAddress   +
                               (int)sh.PointerToRawData +
                               (int)lpFile)    -
                      (int)sh.VirtualAddress   +
                      (int)sh.PointerToRawData +
                      (int)lpFile);

    /* Figure out how much memory to allocate for all strings. */
    pCnt = pNames;
    for (i=0; i<(int)ped->NumberOfNames; i++)
        while (*pCnt++);
    nCnt = (int)(pCnt. pNames);

    /* Allocate memory off heap for function names. */
    *pszFunctions = HeapAlloc (hHeap, HEAP_ZERO_MEMORY, nCnt);

    /* Copy all strings to buffer. */
    CopyMemory ((LPVOID)*pszFunctions, (LPVOID)pNames, nCnt);

    return nCnt;
}
```

Notice that in this function the variable *pNames* is assigned by determining first the address of the offset and then the actual offset location. Both the address of the offset and the offset itself are relative virtual addresses and must be translated before being used, as the function demonstrates. You could write a similar function to determine the ordinal values or entry points of the functions.

**Import data section, .idata**

The .idata section is import data, including the import directory and import address name table. Although an IMAGE_DIRECTORY_ENTRY_IMPORT directory is defined, no corresponding import directory structure is included in the file WINNT.H. Instead, there are several other structures called IMAGE_IMPORT_BY_NAME, IMAGE_THUNK_DATA, and IMAGE_IMPORT_DESCRIPTOR. Personally, I couldn't make heads or tails of how these structures are supposed to correlate to the .idata section, so I spent several hours deciphering the .idata section body and came up with a much simpler structure. I named this structure **IMAGE_IMPORT_MODULE_DIRECTORY**.

```
typedef struct tagImportDirectory
    {
    DWORD    dwRVAFunctionNameList;
    DWORD    dwUseless1;
    DWORD    dwUseless2;
    DWORD    dwRVAModuleName;
    DWORD    dwRVAFunctionAddressList;
    }IMAGE_IMPORT_MODULE_DIRECTORY,
     * PIMAGE_IMPORT_MODULE_DIRECTORY;
```

Unlike the data directories of other sections, this one repeats one after another for each imported module in the file. Think of it as an entry in a list of module data directories, rather than a data directory to the entire section of data. Each entry is a directory to the import information for a specific module.

One of the fields in the **IMAGE_IMPORT_MODULE_DIRECTORY** structure is *dwRVAModuleName*, a relative virtual address pointing to the name of the module. There are also two *dwUseless* parameters in the structure that serve as padding to keep the structure aligned properly within the section. The PE file format specification mentions something about import flags, a time/date stamp, and major/minor versions, but these two fields remained empty throughout my experimentation, so I still consider them useless.

Based on the definition of this structure, you can retrieve the names of modules and all functions in each module that are imported by an executable file. The following function demonstrates how to retrieve all the module names imported by a particular PE file:

```
int  WINAPI GetImportModuleNames (
    LPVOID    lpFile,
    HANDLE    hHeap,
    char      **pszModules)
{
    PIMAGE_IMPORT_MODULE_DIRECTORY  pid;
    IMAGE_SECTION_HEADER            idsh;
    BYTE                            *pData;
    int                             nCnt = 0, nSize = 0, i;
    char                            *pModule[1024];
    char                            *psz;

    pid = (PIMAGE_IMPORT_MODULE_DIRECTORY)ImageDirectoryOffset
             (lpFile, IMAGE_DIRECTORY_ENTRY_IMPORT);
    pData = (BYTE *)pid;

    /* Locate section header for ".idata" section. */
    if (!GetSectionHdrByName (lpFile, &idsh, ".idata"))
        return 0;

    /* Extract all import modules. */
    while (pid->dwRVAModuleName)
        {
        /* Allocate buffer for absolute string offsets. */
        pModule[nCnt] = (char *)(pData +
               (pid->dwRVAModuleName-idsh.VirtualAddress));
        nSize += strlen (pModule[nCnt]) + 1;

        /* Increment to the next import directory entry. */
        pid++;
        nCnt++;
        }

    /* Copy all strings to one chunk of heap memory. */
    *pszModules = HeapAlloc (hHeap, HEAP_ZERO_MEMORY, nSize);
    psz = *pszModules;
    for (i=0; i<nCnt; i++)
        {
        strcpy (psz, pModule[i]);
        psz += strlen (psz) + 1;
        }

    return nCnt;
}
```

The function is pretty straightforward. However, one thing is worth pointing out--notice the while loop. This loop is terminated when *pid->dwRVAModuleName* is 0. Implied here is that at the end of the list of **IMAGE_IMPORT_MODULE_DIRECTORY** structures is a null structure that has a value of 0 for at least the *dwRVAModuleName* field. This is the behavior I observed in my experimentation with the file and later confirmed in the PE file format specification.

The first field in the structure, *dwRVAFunctionNameList*, is a relative virtual address to a list of relative virtual addresses that each point to the function names within the file. As shown in the following data, the module and function names of all imported modules are listed in the .idata section data:

```
E6A7 0000 F6A7 0000  08A8 0000 1AA8 0000  ................
28A8 0000 3CA8 0000  4CA8 0000 0000 0000  (...<...L.......
0000 4765 744F 7065  6E46 696C 654E 616D  ..GetOpenFileNam
6541 0000 636F 6D64  6C67 3332 2E64 6C6C  eA..comdlg32.dll
0000 2500 4372 6561  7465 466F 6E74 496E  ..%.CreateFontIn
6469 7265 6374 4100  4744 4933 322E 646C  directA.GDI32.dl
6C00 A000 4765 7444  6576 6963 6543 6170  l...GetDeviceCap
7300 C600 4765 7453  746F 636B 4F62 6A65  s...GetStockObje
6374 0000 D500 4765  7454 6578 744D 6574  ct....GetTextMet
```

```
7269 6373 4100 1001  5365 6C65 6374 4F62  ricsA...SelectOb
6A65 6374 0000 1601  5365 7442 6B43 6F6C  ject....SetBkCol
6F72 0000 3501 5365  7454 6578 7443 6F6C  or..5.SetTextCol
6F72 0000 4501 5465  7874 4F75 7441 0000  or..E.TextOutA..
```

This particular section represents the beginning of the list of import module and function names. If you begin examining the right section part of the data, you should recognize the names of familiar Win32 API functions and the module names they are found in. Reading from the top down, you get **GetOpenFileNameA**, followed by the module name COMDLG32.DLL. Shortly after that, you get **CreateFontIndirectA**, followed by the module GDI32.DLL and then the functions **GetDeviceCaps**, **GetStockObject**, **GetTextMetrics**, and so forth.

This pattern repeats throughout the .idata section. The first module name is COMDLG32.DLL and the second is GDI32.DLL. Notice that only one function is imported from the first module, while many functions are imported from the second module. In both cases, the function names and the module name to which they belong are ordered such that a function name appears first, followed by the module name and then by the rest of the function names, if any.

The following function demonstrates how to retrieve the function names for a specific module:

```
int  WINAPI GetImportFunctionNamesByModule (
    LPVOID    lpFile,
    HANDLE    hHeap,
    char      *pszModule,
    char      **pszFunctions)
{
    PIMAGE_IMPORT_MODULE_DIRECTORY  pid;
    IMAGE_SECTION_HEADER     idsh;
    DWORD                    dwBase;
    int                      nCnt = 0, nSize = 0;
    DWORD                    dwFunction;
    char                     *psz;


    /* Locate section header for ".idata" section. */
    if (!GetSectionHdrByName (lpFile, &idsh, ".idata"))
        return 0;

    pid = (PIMAGE_IMPORT_MODULE_DIRECTORY)ImageDirectoryOffset
            (lpFile, IMAGE_DIRECTORY_ENTRY_IMPORT);

    dwBase = ((DWORD)pid. idsh.VirtualAddress);

    /* Find module's pid. */
    while (pid->dwRVAModuleName &&
           strcmp (pszModule,
                  (char *)(pid->dwRVAModuleName+dwBase)))
        pid++;

    /* Exit if the module is not found. */
    if (!pid->dwRVAModuleName)
        return 0;

    /* Count number of function names and length of strings. */
    dwFunction = pid->dwRVAFunctionNameList;
    while (dwFunction                      &&
           *(DWORD *)(dwFunction + dwBase) &&
           *(char *)((*(DWORD *)(dwFunction + dwBase)) +
            dwBase+2))
        {
        nSize += strlen ((char *)((*(DWORD *)(dwFunction +
             dwBase)) + dwBase+2)) + 1;
        dwFunction += 4;
        nCnt++;
        }

    /* Allocate memory off heap for function names. */
    *pszFunctions = HeapAlloc (hHeap, HEAP_ZERO_MEMORY, nSize);
    psz = *pszFunctions;

    /* Copy function names to memory pointer. */
    dwFunction = pid->dwRVAFunctionNameList;
    while (dwFunction                      &&
           *(DWORD *)(dwFunction + dwBase) &&
           *((char *)((*(DWORD *)(dwFunction + dwBase)) +
            dwBase+2)))
        {
        strcpy (psz, (char *)((*(DWORD *)(dwFunction + dwBase)) +
                dwBase+2));
        psz += strlen((char *)((*(DWORD *)(dwFunction + dwBase))+
                dwBase+2)) + 1;
        dwFunction += 4;
        }

    return nCnt;
}
```

Like the **GetImportModuleNames** function, this function relies on the end of each list of information to have a zeroed entry. In this case, the list of function names ends with one that is zero.

The final field, *dwRVAFunctionAddressList*, is a relative virtual address to a list of virtual addresses that will be placed in the section data by the loader when the file is loaded. Before the file is loaded, however, these virtual addresses are replaced by relative virtual addresses that correspond exactly to the list of function names. So before the file is loaded, there are two identical lists of relative virtual addresses pointing to imported function names.

### Debug information section, .debug

Debug information is initially placed in the .debug section. The PE file format also supports separate debug files (normally identified with a .DBG extension) as a means of collecting debug information in a central location. The debug section contains the debug information, but the debug directories live in the .rdata section mentioned earlier. Each of those directories references debug information in the .debug section. The debug directory structure is defined as an **IMAGE_DEBUG_DIRECTORY**, as follows:

**WINNT.H**

```
typedef struct _IMAGE_DEBUG_DIRECTORY {
    ULONG   Characteristics;
    ULONG   TimeDateStamp;
    USHORT  MajorVersion;
    USHORT  MinorVersion;
    ULONG   Type;
    ULONG   SizeOfData;
    ULONG   AddressOfRawData;
    ULONG   PointerToRawData;
} IMAGE_DEBUG_DIRECTORY, *PIMAGE_DEBUG_DIRECTORY;
```

The section is divided into separate portions of data representing different types of debug information. For each one there is a debug directory described above. The different types of debug information are listed below:

**WINNT.H**

```
#define IMAGE_DEBUG_TYPE_UNKNOWN          0
#define IMAGE_DEBUG_TYPE_COFF             1
#define IMAGE_DEBUG_TYPE_CODEVIEW         2
#define IMAGE_DEBUG_TYPE_FPO              3
#define IMAGE_DEBUG_TYPE_MISC             4
```

The *Type* field in each directory indicates which type of debug information the directory represents. As you can see in the list above, the PE file format supports many different types of debug information, as well as some other informational fields. Of those, the **IMAGE_DEBUG_TYPE_MISC** information is unique. This information was added to represent miscellaneous information about the executable image that could not be added to any of the more structured data sections in the PE file format. This is the only location in the image file where the image name is sure to appear. If an image exports information, the export data section will also include the image name.

Each type of debug information has its own header structure that defines its data. Each of these is listed in the file WINNT.H. One nice thing about the **IMAGE_DEBUG_DIRECTORY** structure is that it includes two fields that identify the debug information. The first of these, *AddressOfRawData*, is the relative virtual address of the data once the file is loaded. The other, *PointerToRawData*, is an actual offset within the PE file, where the data is located. This makes it easy to locate specific debug information.

As a last example, consider the following function, which extracts the image name from the **IMAGE_DEBUG_MISC** structure:

```
int    WINAPI RetrieveModuleName (
    LPVOID    lpFile,
    HANDLE    hHeap,
    char      **pszModule)
{

    PIMAGE_DEBUG_DIRECTORY    pdd;
    PIMAGE_DEBUG_MISC         pdm = NULL;
    int                       nCnt;

    if (!(pdd = (PIMAGE_DEBUG_DIRECTORY)ImageDirectoryOffset
               (lpFile, IMAGE_DIRECTORY_ENTRY_DEBUG)))
        return 0;

    while (pdd->SizeOfData)
        {
        if (pdd->Type == IMAGE_DEBUG_TYPE_MISC)
            {
            pdm = (PIMAGE_DEBUG_MISC)
                ((DWORD)pdd->PointerToRawData + (DWORD)lpFile);

            nCnt = lstrlen (pdm->Data)*(pdm->Unicode?2:1);
            *pszModule = (char *)HeapAlloc (hHeap,
                                            HEAP_ZERO_MEMORY,
                                            nCnt+1;
            CopyMemory (*pszModule, pdm->Data, nCnt);

            break;
            }

        pdd ++;
        }

    if (pdm != NULL)
        return nCnt;
    else
        return 0;
}
```

As you can see, the structure of the debug directory makes it relatively easy to locate a specific type of debug information. Once the **IMAGE_DEBUG_MISC** structure is located, extracting the image name is as simple as invoking the **CopyMemory** function.

As mentioned above, debug information can be stripped into separate .DBG files. The Windows NT SDK includes a utility called REBASE.EXE that serves this purpose. For example, in the following statement an executable image named TEST.EXE is being stripped of debug information:

```
rebase -b 40000 -x c:\samples\testdir test.exe
```

The debug information is placed in a new file called TEST.DBG and located in the path specified, in this case c:\samples\testdir. The file begins with a single **IMAGE_SEPARATE_DEBUG_HEADER** structure, followed by a copy of the section headers that exist in the stripped executable image. Then the .debug section data follows the section headers. So, right after the section headers are the series of **IMAGE_DEBUG_DIRECTORY** structures and their associated data. The debug information itself retains the same structure as described above for normal image file debug information.